

Wilkinson's Inertia-Revealing Factorization and Its Application to Sparse Matrices

Alex Druinsky*
LBNL

Eyal Carlebach*
Verint Systems

Sivan Toledo
Tel Aviv University

November 2015

Abstract

We propose a new inertia-revealing factorization for sparse matrices. The factorization scheme and the method for extracting the inertia from it were proposed in the 1960s for dense, banded, or tridiagonal matrices, but they have been abandoned in favor of faster methods. We show that this scheme can be applied to any sparse matrix and that the fill in the factorization is bounded by the fill in the sparse QR factorization of the same matrix (but is usually much smaller). We present experimental results, studying the method's numerical stability and performance. Our implementation of the method is somewhat naive, but still demonstrates its potential.

1 Introduction

The inertia of a real symmetric matrix A is a triplet of numbers that count how many of A 's eigenvalues are positive, negative or zero, respectively. The first two of these numbers are called the positive and negative indices of inertia, respectively, and the last one is the nullity of the matrix. The inertia is easy to obtain by computing all of the eigenvalues of A . However, even for moderately large sparse problems, computing all of the eigenvalues is either expensive or infeasible. Over the years, inertia algorithms have been developed that do not resort to computing the full spectrum. Such algorithms are the focus of this paper.

A fast inertia algorithm can facilitate a number of computational tasks. As a first example, consider an eigenvalue of A that is isolated from the rest of the spectrum within a known interval (x_0, x_1) . Let us see how to compute the multiplicity of that eigenvalue by taking advantage of a fast inertia algorithm. The key is the property that the negative index of inertia of the matrix $A - xI$, where x is real, is the number of eigenvalues of A that lie to the left of x . To compute the multiplicity, we compute the negative indices of inertia of the

*The first two authors carried this work out while they were students at Tel Aviv University.

matrices $A - x_1I$ and $A - x_0I$, and subtract the latter index from the former. This yields the number of eigenvalues that are to the left of x_1 but not to the left of x_0 , which is exactly the multiplicity of the target eigenvalue.

The same idea works if we need to compute the aggregate multiplicity of all of the eigenvalues within a known interval. This is useful when we compute all of the eigenvalues within that interval using a Lanczos shift-and-invert eigensolver, and we wish to ascertain that no eigenvalues have been missed [48].

Another example is when we need to compute a histogram of the spectrum within an interval of interest. In this case, we divide the interval into subintervals and compute the number of eigenvalues within each subinterval using the above technique. Such a histogram is necessary when we wish to compute all of the eigenvalues within the interval of interest, and we need to partition the work evenly among multiple processors [1]. Here, each inertia computation is independent from the others, which makes this scheme suitable for parallel computation.

The final example is bisection. In bisection, we compute the eigenvalues of A by using the negative index of inertia of $A - xI$ as a binary-search oracle for locating the eigenvalues by their ordinates (for an example, see [23, Section 8.4.1]).

To facilitate the above (and similar) tasks, we propose a new algorithm that computes the inertia of sparse symmetric matrices. Our new sparse inertia algorithm combines two old and obsolete algorithmic ideas into a useful and novel algorithm. One idea is an algorithm that was proposed by Wilkinson for computing the inertia of dense, banded, or tridiagonal matrices [47, pp. 236–240]. Wilkinson recommended the algorithm as the building block of a bisection eigensolver for tridiagonal matrices. Over the years, this algorithm was replaced by the use of LDL^T factorizations of shifts of the matrix (and by other tridiagonal eigensolvers; see [11, 41, 46]). The newer alternatives are more efficient and more stable; Wilkinson’s inertia idea appeared to be obsolete. The other idea is the row-by-row sparse QR factorization by George and Heath [18]. It was replaced by sparse multifrontal QR factorizations [37] (for a recent implementation, see [8]). We have discovered that George and Heath’s approach to sparse factorizations and its analysis can be applied to Wilkinson’s inertia algorithm. This discovery is what motivates the new algorithm.

Existing algorithms for computing the inertia of sparse matrices work by computing a so-called symmetric-indefinite factorization (see [14] for an early example). Such algorithms work well in practice, but they are heuristic in the sense that no guarantees can be made about the amount of work and memory that they require on any individual matrix. In contrast, on many important classes of matrices, the new algorithm provides an a priori bound on the fill that it generates and hence on its storage requirements and (indirectly) on its running time. A counterintuitive feature of the algorithm is that it computes a nonsymmetric factorization; we sacrifice symmetry, but we gain bounds on the fill.

Our implementation of the method is inspired by CSpase, Tim Davis’s concise sparse-matrix library [7]. We felt that it was premature to invest the kind of years-long engineering effort that is required to produce production-

quality sparse-matrix factorization software, and so we translated the method into code in a straightforward way, opting for simplicity whenever possible. This means that the resulting code does not provide optimal performance, but its performance is predictable and easy to understand to anyone who understands the basic algorithm.

The rest of the paper is organized as follows. We present the required background material in Section 2. Section 3 describes our sparse implementation and its effect on the sparsity pattern of the matrix. We analyze the numerical stability of the algorithm in Section 4, and present numerical experiments in Section 5. The performance of our implementation is studied experimentally in Section 6, followed by our conclusions from this research in Section 7.

2 Background

2.1 General-Purpose Eigensolvers

Eigenvalues of symmetric dense matrices are almost always computed using a two-step procedure [23, Section 8]. In the first step, we reduce the matrix to tridiagonal form using a sequence of orthogonal similarity transformations. This reduction has the form $Q^T A Q = T$, where Q is orthogonal and T is tridiagonal. The reduction preserves the spectrum of A and allows us to compute A 's eigenvalues by applying a tridiagonal eigensolver to T ; this is the second step.

The computational cost of reducing an n -by- n matrix A to T is $\Theta(n^3)$ arithmetic operations, and the cost of tridiagonal eigensolvers such as bisection, the implicit QR method or the more recent MRRR algorithm [13] is $\Theta(n^2)$. Because of the upfront investment in reducing the matrix to T , there is no advantage in computing a subset of the eigenvalues; the whole spectrum is produced at essentially the same cost. Eigenvectors can also be computed: this costs $\Theta(n^2)$ for the eigenvectors of T and $\Theta(n^3)$ to transform them into eigenvectors of A by reversing the effect of the orthogonal similarities.

2.2 Lanczos

The orthogonal-tridiagonalization approach is not suitable for sparse matrices because orthogonal similarity transformations destroy the matrix's sparsity, resulting in a computational cost of $\Theta(n^3)$ even when A is very sparse. The most widely applicable approach for sparse symmetric eigenproblems is the Lanczos iteration. Here we compute an orthogonal basis Q of the Krylov subspace $K_t = \text{span}(b, Ab, \dots, A^{t-1}b)$, where t is the dimension of the subspace and b is a nonzero starting vector, usually chosen at random. We then use Q to orthogonally project A along the Krylov subspace, producing a t -by- t tridiagonal matrix T whose eigenvalues serve as approximations to the eigenvalues of A . A popular implementation of this approach is the implicitly restarted Lanczos method, available in the ARPACK library [4, 34].

Lanczos produces each column of Q by multiplying A with the previous column, and then orthogonalizing the resulting vector against the existing columns. The computational cost is $t - 1$ matrix-vector products and $\Theta(t^2n)$ arithmetic operations for the orthogonalization operations. Because of the steep dependence on t , the number of iterative steps is usually chosen to be much smaller than n . Stopping after a small number of iterations means that only a few of A 's eigenvalues are well approximated, typically those at the edges of the spectrum.

When we wish to compute eigenvalues in the interior of the spectrum, we compute a triangular factorization of $A - \sigma I$, where σ is the point of interest in the spectrum, and iterate with $(A - \sigma I)^{-1}$ instead of iterating with A . This transformation brings out the interior eigenvalues that are closest to σ toward the edges of the spectrum, which leads Lanczos to converge on them. This is called shift-and-invert Lanczos.

The drawback of Lanczos is that it is hard to use as a black-box solver. Convergence depends on the structure of the spectrum, and there is no way to ascertain that all of the required eigenvalues have been obtained. Another difficulty stems from the fact that the projection of K_t along any eigenspace of A is one dimensional, and therefore Lanczos cannot determine the multiplicities of the eigenvalues of A . If we wish to determine multiplicities, we can use a variant called block Lanczos, but the costs rise with the dimension of the computed eigenspace, which limits the efficiency of this approach.

2.3 Bisection

Bisection is a recursive algorithm that computes all of the eigenvalues that lie in a user-specified half-open interval $[x_0, x_1)$. Since computing an interval that envelops the entire spectrum is relatively easy (e.g., using the bound $|\lambda| \leq \|A\|_2 \leq \|A\|_1$; see [23, Theorem 7.2.1]), the requirement for a user-provided interval does not limit the generality of the method. Bisection works by splitting the interval into two halves, counting the eigenvalues in each half, and continuing the search recursively in each half that contains eigenvalues. To count the eigenvalues in an interval $[x_0, x_1)$, we determine the numbers $f(x_0)$ and $f(x_1)$ of eigenvalues that lie strictly to the left of x_0 and of x_1 ; the difference $f(x_1) - f(x_0)$ is the number of eigenvalues in the interval. The function $f(x)$ is evaluated by computing the negative index of inertia ν of $A - xI$,

$$f(x) = \nu(A - xI).$$

The recursion stops when the length $x_1 - x_0$ of the current interval falls below a user-specified tolerance threshold. When the algorithm returns, it reports all of the intervals at or below the length threshold and the number of eigenvalues in each such interval (numbers greater than one may correspond to a tight cluster of distinct eigenvalues or to multiple eigenvalues).

A pseudocode of bisection is given in Algorithm 1. The code can be easily modified to compute only the eigenvalues inside a user-specified interval or a set of eigenvalues that are specified by their ordinals.

Algorithm 1 Computing all of the eigenvalues using bisection.

```

1: Input: A symmetric matrix  $A \in \mathbb{R}^{n \times n}$  and the tolerance threshold  $\tau$ .
2: Output: A vector of eigenvalues  $e \in \mathbb{R}^n$ .
3:
4: function BISECTION
5:    $x_0 \leftarrow -\|A\|_1$ 
6:    $\nu_0 \leftarrow 0$ 
7:    $x_1 \leftarrow \|A\|_1$ 
8:    $\nu_1 \leftarrow n$ 
9:    $e \leftarrow \text{BISECT}(x_0, \nu_0, x_1, \nu_1)$ 
10:  return  $e$ 
11: end function
12:
13: function BISECT( $x_0, \nu_0, x_1, \nu_1$ )
14:    $x \leftarrow 0.5(x_0 + x_1)$ 
15:   if  $x_1 - x_0 > \tau \|A\|_1$  then
16:      $\mu \leftarrow \nu(A - xI)$   $\triangleright$  Compute the negative index of inertia of  $A - xI$ .
17:     if  $\mu > \nu_0$  then
18:        $e_0 \leftarrow \text{BISECT}(x_0, \nu_0, x, \mu)$ 
19:     end if
20:     if  $\nu_1 > \mu$  then
21:        $e_1 \leftarrow \text{BISECT}(x, \mu, x_1, \nu_1)$ 
22:     end if
23:     return  $[e_0, e_1]$ 
24:   else
25:     return the eigenvalue  $x$  with multiplicity  $\nu_1 - \nu_0$ 
26:   end if
27: end function

```

The computational cost of bisection is as follows. Every node of the recursion tree requires computing $\nu(A - xI)$, whose cost we denote by $N(A)$. The number of nodes in the tree is bounded by the number of leaves times the height of the tree; the number of leaves equals the number of eigenvalues k that we compute and the height of the tree equals $\Theta(\log(1/\tau))$, where τ is the error tolerance threshold. Therefore the total cost is $O(k \log(1/\tau)N(A))$.

As we explained in Section 2.1, we compute the eigenvalues of a dense matrix by first reducing it orthonormally to a tridiagonal one. This allows us to compute $\nu(A - xI)$ using an LDL^T factorization without pivoting at an arithmetic cost of $O(n)$ operations. Using the LDL^T factorization without pivoting to compute the inertia is in general not numerically stable. However, for a narrow class of matrices that includes the tridiagonal ones, this method is guaranteed to work [12]. In LAPACK, it is implemented in the subroutine STEBZ [2, 32]. This approach gives a total arithmetic cost of $\Theta(n^3 + k \log(1/\tau)n)$, where the n^3 term is the cost of the tridiagonal reduction.

When the matrix is sparse, we must either compute an LDL^T factorization with pivoting using a sparse symmetric-indefinite factorization algorithm (several such algorithms exist; we address them next), or use our new algorithm. In either case, bisection is not competitive with shift-and-invert Lanczos for the canonical problem of accurately computing the eigenvalues that are closest to a user-selected point of interest. For this problem, Lanczos requires a single sparse factorization, while bisection requires a sequence of $\Omega(\log(1/\tau))$, even when computing a single eigenvalue. Nevertheless, bisection has two advantages that make it potentially useful when the required accuracy is moderate. First, bisection decouples the work that takes place within disjoint intervals of the spectrum, making it suitable for parallel computation, and second, bisection can locate eigenvalues by their ordinals, which is not possible with other eigensolvers.

2.4 Sparse Symmetric-Indefinite Factorizations

An alternative to our sparse inertia algorithm is to use a sparse symmetric-indefinite factorization. Such factorizations have the form $PAP^T = LDL^T$, where P is a permutation matrix and D is a block-diagonal one with 1-by-1 and 2-by-2 diagonal blocks. The inertia of D is trivial to compute because each of its diagonal blocks corresponds to one or two easy-to-compute eigenvalues. By Sylvester's Law of Inertia (see [23, Theorem 8.1.17]), the matrices D and A have the same inertia, and therefore having D allows us to easily compute the inertia of A . Examples of modern symmetric-indefinite factorization codes include MA86 and MA57 from the HSL library [29] and the sparse symmetric-indefinite solver from the PARDISO library [43].

2.5 Sparse Factorizations and Fill Bounds

In some sparse factorizations, the fill in the factors has a tight bound that is fully determined by the sparsity pattern of A . This is the case for the Cholesky

factorization $A = LL^T$ of a symmetric positive-definite matrix, and for the LU factorization with partial pivoting $QA = LU$. When such a bound exists, one can preorder the matrix to reduce fill. For example, the sparse Cholesky factorization is typically applied to PAP^T , and the sparse LU to AP^T , where P is a permutation chosen to minimize fill (perhaps heuristically). Even when P is selected heuristically, once it is fixed, the bound on the fill is determined and can be easily computed.

This is not the case for numerically stable sparse symmetric-indefinite factorizations, which include MA86, MA57 and PARDISO. The type of pivoting that they perform can destroy any a priori bound on the fill. These factorizations are typically applied to PAP^T where P is chosen optimistically so as to reduce fill assuming that pivoting will not change the fill. When this assumption is true, or when pivoting generates just a bit more fill than would be generated without pivoting, these factorizations are very effective. But there is no sparse symmetric-indefinite algorithm that provides a guaranteed a priori bound on the fill, and as we shall see below, such algorithms do sometimes experience catastrophic amounts of fill.

On the other hand, the fill in our new algorithm *can* be bounded a priori. This bound is derived from the bound on fill in the sparse QR factorization, which we now explain. But to explain fill in QR , we need to first explain fill in Cholesky, which is best described in terms of the graph (V_A, E_A) of the symmetric matrix A . The vertex set is simply $\{1, 2, \dots, n\}$, representing the rows and columns of the matrix, and the edge set contains the (i, j) pairs that correspond to the nonzero elements of A . A seminal (and fairly easy) result shows that if $L_{ij} \neq 0$, then there is a path in the graph from i to j consisting entirely of vertices with indices smaller than i and j [42]. This is a necessary condition. It is normally also sufficient; only exact cancellation can lead to $L_{ij} = 0$ in the presence of such a path.

The key to provable fill minimization in Cholesky is a method called nested dissection [17] (see also [7, Section 7.6]). Nested dissection generates a good permutation P by finding a small set of vertices called an *approximately balanced vertex separator*, ordering that set last, and recursing on two subgraphs. This method provides powerful guarantees on the fill of sparse Cholesky. For example, if the graph of A is a \sqrt{n} -by- \sqrt{n} finite-element mesh or a planar graph, ordering by nested dissection guarantees that the Cholesky factor contains $O(n \log n)$ nonzero elements and the factorization requires $O(n^{3/2})$ arithmetic operations. More general bounds for matrices whose graphs have hierarchies of small separators have also been proved [36].

The R factor of the QR factorization of A is also the Cholesky factor of $A^T A$, and therefore the fill of the R factor of a sparse matrix A can be controlled by computing a nested-dissection ordering for $A^T A$ and reordering the columns of A accordingly. George and Ng have analyzed this and proved that the sparse QR factorization of a possibly unsymmetric or indefinite \sqrt{n} -by- \sqrt{n} finite-element mesh has the same asymptotic costs as the Cholesky factorization of a symmetric positive-definite matrix of this type [19]. Furthermore, a nested-dissection ordering of $A^T A$ can be computed efficiently without computing $A^T A$ or even

its sparsity structure. Recent algorithms that compute such orderings include hypergraph-partitioning-based nested dissection, wide-separator nested dissection and reduction to singly bordered block-diagonal form [3, 24, 30].

2.6 Wilkinson's Inertia Algorithm

We base our algorithm on a method that Wilkinson proposed for computing the inertia of general matrices [47, p. 236–240], which has also been adapted for banded matrices [26, 27, 39, 44]. The method is based on the Sturm sequence property, which is described by the following theorem. The theorem states that the number of negative eigenvalues of a matrix can be computed by forming the sequence of the determinants of its leading principal minors and counting the number of sign changes between consecutive elements of that sequence. The proof of the theorem is based on the eigenvalue interlacing property (for details, see [47, p. 300]).

Theorem 1 (Sturm Sequence Property). *The number of negative eigenvalues of a symmetric matrix $A \in \mathbb{R}^{n \times n}$ equals the number of sign changes between consecutive elements of the sequence*

$$1, \det(A_1), \det(A_2), \dots, \det(A_n),$$

where $A_k = A_{1:k, 1:k}$ are the leading principal submatrices of A for $k = 1, 2, \dots, n$, assuming that none of the elements of that sequence are zero.

We can, therefore, compute the number of negative eigenvalues by forming the sequence of the determinants of the leading principal minors. Wilkinson proposed to form this sequence by reducing the matrix to upper-triangular form one row after another, and producing the determinant of the corresponding leading principal minor as a byproduct after processing each row. Let us describe this idea in concrete form using matrix notation. The first k steps of the algorithm reduce the first k rows of the matrix to a k -by- n upper-trapezoidal matrix $U^{(k)}$. The transformation that we apply to $A_{1:k, :}$ in these steps can be described in aggregate using a k -by- k matrix $X^{(k)}$, which is not formed by the algorithm and is computed only implicitly. Using this notation, we may write

$$X^{(k)} A_{1:k, :} = U^{(k)}.$$

Let us take the determinant of the leading k -by- k principal submatrix on both sides of this equation. This yields

$$\det(X^{(k)} A_{1:k, 1:k}) = \det(U_{1:k, 1:k}^{(k)}),$$

which we may write as

$$\det(X^{(k)}) \det(A_k) = u_{11}^{(k)} u_{22}^{(k)} \cdots u_{kk}^{(k)},$$

where $u_{11}^{(k)}, u_{22}^{(k)}, \dots, u_{kk}^{(k)}$ are the diagonal elements of $U^{(k)}$. This gives

$$\det(A_k) = \frac{u_{11}^{(k)} u_{22}^{(k)} \cdots u_{kk}^{(k)}}{\det(X^{(k)})}.$$

Computing the numerator in the above formula is trivial. To explain how we compute $\det(X^{(k)})$, we need to explain how $U^{(k)}$ is produced from $U^{(k-1)}$.

Let the reduced matrix $A^{(k)}$ that we obtain from step k be defined as the matrix that contains the rows of the upper-trapezoidal $U^{(k)}$, followed by the yet-unprocessed rows of A . For $n = 6$ and $k = 4$, this matrix has the form

$$A^{(k)} = \begin{matrix} & U^{(k)} \\ A_{(k+1):n,:} & \left[\begin{array}{cccccc} \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times \\ & & \times & \times & \times & \times \\ & & & \times & \times & \times \\ \hline \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \end{array} \right] \end{matrix}.$$

For completeness, we also define that $A^{(0)} = A$. To compute $U^{(k)}$, we eliminate the elements in row k of $A^{(k-1)}$ one by one from left to right using a sequence of elimination operations, stopping when we reach the diagonal element. We can choose between two types of elimination operations: elementary stabilized transformations and Givens rotations. An elementary stabilized transformation works by eliminating an element in position (k, j) in two steps:

1. Test whether $|A_{jj}| < |A_{kj}|$, and if so, interchange rows k and j .
2. Subtract a multiple of row j from row k using A_{kj}/A_{jj} as the scaling factor.

A Givens rotation works by premultiplying rows k and j with the rotation matrix

$$G^{(k,j)} = \frac{1}{\sqrt{A_{jj}^2 + A_{kj}^2}} \begin{bmatrix} A_{jj} & A_{kj} \\ -A_{kj} & A_{jj} \end{bmatrix}.$$

(Here, by A_{kj} and A_{jj} we refer to the elements of the reduced matrix on which we are operating, not those of the original A .)

Having completed the $k-1$ elimination operations (if A is sparse, fewer than $k-1$ may be necessary), we can now compute $\det(A_k)$. If we use Givens rotations, $\det(X^{(k)}) = 1$. The determinant of a rotation matrix always equals 1, the determinant of a product of rotations is therefore also 1, and so $\det(X^{(k)}) = 1$. If we use elementary stabilized transformations, $\det(X^{(k)}) = \pm 1$, and resolving the sign is straightforward. In every elimination step, we perform up to two operations: an interchange of a pair of rows, which corresponds to a permutation matrix with determinant -1 , and a subtraction of a multiple of one row from a subsequent row, which corresponds to a unit lower-triangular matrix whose determinant is 1. Therefore, when using elementary stabilized transformations, $\det(X^{(k)})$ is either 1 or -1 , depending on whether the number of row interchanges that we have made is even or odd.

For completeness, we list a pseudocode of the variant of the algorithm that uses elementary stabilized transformations in Algorithm 2. In the pseudocode, we simplified the algorithm by counting the number of sign changes in the determinant sequence without actually computing the determinants.

Algorithm 2 Computing the number of negative eigenvalues using elementary stabilized transformations.

```

1: Input: a symmetric matrix  $A \in \mathbb{R}^{n \times n}$ .
2: Output: the number  $\nu$  of negative eigenvalues of  $A$ .
3:
4:  $\nu \leftarrow 0$ 
5: for  $i \leftarrow 1, 2, \dots, n$  do
6:    $x \leftarrow 0$  ▷ number of row interchanges and diagonal sign changes
7:   for  $j \leftarrow 1, 2, \dots, i - 1$  do
8:     if  $|A_{jj}| < |A_{ij}|$  then
9:        $A_{j,:} \leftrightarrow A_{i,:}$ 
10:       $x \leftarrow x + 1$ 
11:      if  $\text{sign}(A_{jj}) \neq \text{sign}(A_{ij})$  then
12:         $x \leftarrow x + 1$ 
13:      end if
14:    end if
15:     $A_{i,:} \leftarrow A_{i,:} - \frac{A_{ij}}{A_{jj}} A_{j,:}$ 
16:  end for
17:  if  $A_{ii} < 0$  then
18:     $x \leftarrow x + 1$ 
19:  end if
20:  if  $x \bmod 2 = 1$  then
21:     $\nu \leftarrow \nu + 1$ 
22:  end if
23: end for

```

3 The New Sparse Inertia Algorithm

3.1 Bounds on Fill and Arithmetic Cost

In the Givens rotations-based variant of the factorization, the computation is identical to that of George and Heath's sparse QR factorization, which factors the matrix row by row [18]. The fill in this case is obviously identical to that of sparse QR , and the number of arithmetic operations is the same as in George and Heath's algorithm.

If we use elementary stabilized transformations in Wilkinson's algorithm, the fill that the algorithm produces is bounded by the fill in George and Heath's QR factorization.

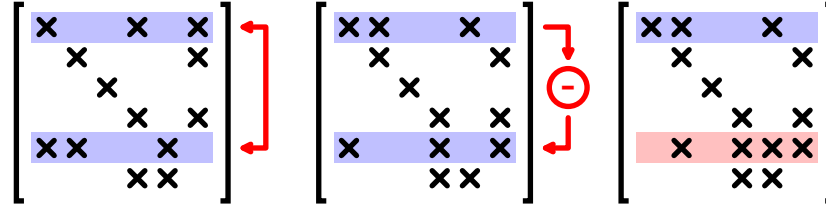
Definition 2. Element $A_{ij}^{(k)}$ in position i, j of the k th reduced matrix in some factorization algorithm is called a *structural nonzero* if it is different from zero whenever the algorithm is carried out in an arithmetic that satisfies the following for all scalars x and y :

1. If $x + y = 0$ or $x - y = 0$, then $x = y = 0$.
2. $0x = 0$.

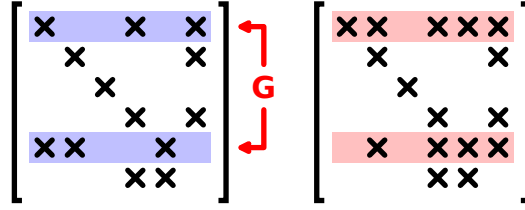
Theorem 3. If $A_{ij}^{(k)}$ is a structural nonzero in the variant of Wilkinson's algorithm that uses elementary stabilized transformations, then $A_{ij}^{(k)}$ is also a structural nonzero in the variant that uses Givens rotations and in George and Heath's QR factorization.

Proof. We prove the result by induction on k . Clearly, the statement is true for $k = 0$, because no elimination steps have taken place. In step k , when an elementary stabilized transformation eliminates A_{ij} , it either leaves the structure of row j unchanged, or it replaces that structure with the one of row i if an interchange of rows is necessary. The structure of row i is then replaced with the union of the two row structures, with the exception of A_{ij} , which becomes zero. In a Givens rotation, the structure of both rows is replaced with their union, again with the exception of A_{ij} . Therefore the sparsity structure that we obtain from a Givens rotation serves as an upper bound on the one that we obtain from an elementary stabilized transformation. Figure 1 illustrates the two types of elimination steps. \square

The stabilized-elementary variant performs fewer arithmetic operations than the QR variant. In the dense setting, elementary stabilized transformations require three times fewer arithmetic operations (because each elementary transformation is cheaper to perform than the corresponding Givens rotation). When the matrix is sparse, the effect can be much more dramatic, because fill that is not created in the lower triangle does not need to be eliminated (there are fewer transformations to apply) and because sparser rows mean that each transformation is cheaper.



(a) Elementary stabilized transformation



(b) Givens rotation

Figure 1: A comparison of the effect that an elementary stabilized transformation has on the sparsity structure to the effect of a Givens rotation. Here, $i = 5$, $j = 1$, and the elementary stabilized transformation exchanges rows i and j before eliminating A_{ij} .

We acknowledge that operation counts are not good predictors of running times on modern computers. To achieve fast running times, it is essential to exploit parallelism and to reduce communication. In sparse factorization, these optimizations are usually performed by exploiting elimination-tree parallelism [31], and by invoking dense subroutines on submatrices that are full or almost full (supernodes; see for example [38]). Our implementation of the algorithm does not currently include such optimizations; implementing them is a significant research/engineering effort which we have not yet undertaken (see, e.g., [6, 15, 35]). Nevertheless, our implementation does demonstrate the sparsity of the algorithm and suggests that a high-performance variant of this algorithm can achieve performance comparable to that of other modern sparse codes.

3.2 A Sparse Implementation

Our implementation of the algorithm follows the outline of Algorithm 2, modified so that we store and operate only on the nonzero elements of the matrix. We represent the matrix using a data structure that we call *expandable compressed sparse row* (ECSR), which is a modification of the widely used compressed sparse row (CSR) layout. (This is an old data structure; see [7, p. 8] for a clear recent exposition.) ECSR exploits the fact that the fill in our algorithm is bounded by the fill in sparse QR , and the fact that the number of structural nonzeros in each row in the R factor can be computed in almost linear time [21]. Our

code uses the function `rowcolcounts` of the CHOLMOD library, version 1.7.0, to compute the row-wise nonzero counts; we call this function using the command `sybifact` in Matlab.

ECSR stores the nonzero elements and their column indices one row after another. The amount of space that we allocate to row i is equal to the computed number of nonzeros in $R_{i,:}$, the i th row in the matrix's R factor (if $A_{i,:}$ is denser than $R_{i,:}$, we allocate enough space for $A_{i,:}$). This guarantees that the space allocated to row i is sufficient to store row i of R and of all the intermediate upper-trapezoidal matrices. The representation consists of the following four vectors:

val numerical values of the nonzeros (in all the rows)
col column indices of the nonzeros (in all the rows)
head pointers into **val** and **col** indicating where each row of the matrix starts
tail pointers indicating where each row ends.

The vacant space of row i is stored in positions **tail**[i] through **head**[$i + 1$].

We use one of two additional data structures to perform row eliminations. In the Givens variant of the algorithm, we use a two-row Sparse Accumulator (SPA) [20], and in the elementary-stabilized variant, we use an Ordered Sparse Accumulator (OSPA) [22]. A multiple-row SPA represents a set of m sparse rows with a shared sparsity structure using an m -by- n matrix **spa.val**, which holds the numerical values of the nonzeros, a length- n vector **spa.occupied** whose elements indicate which columns of **spa.val** are nonzero, and also **spa.col**, a list of the column indices of the nonzeros. The OSPA is similar, except that **spa.col** is stored in a heap data structure [5, Section 6], which provides cheap access to the leftmost nonzero column.

In the Givens variant of the algorithm, we use the SPA to perform Givens rotations as follows.

1. We copy the two rows into the SPA one by one. We start with the nonzero elements of the first row, updating the corresponding elements of **spa.val** and **spa.occupied** and adding the column indices of the nonzeros into **spa.col**. Next we copy the second row in the same way, except that now we use **spa.occupied** to avoid duplicating column indices in **spa.col**.
2. We apply the rotation to each nonzero column, using the column indices in **spa.col** to enumerate the nonzero columns.
3. Finally, we copy the nonzero elements back into the ECSR data structure and restore the SPA to its initial state, using **spa.col** to enumerate elements of **spa.occupied** and **spa.val** that must be cleared.

In this implementation, the total amount of work is proportional to the number of floating-point operations that the Givens rotation performs.

The elementary-stabilized variant is more complex. We implemented it using several subroutines that operate on the ECSR data structure and on a one-row OSPA:

load Loads a row of an ECSR matrix into an empty OSPA. This is similar to loading a row into an unordered SPA, but requires a **build-heap** operation [5, Section 6.3] on the array of column indices.

store Copies the contents of an OSPA back into a row of an ECSR matrix.

retrieve-head Returns the position and the numerical value of the leftmost nonzero element of an OSPA (but leaves it in the OSPA). If the OSPA is empty, returns a special indicator, which we denote \perp .

remove-head Removes the leftmost nonzero element from an OSPA.

subtract Subtracts a multiple of a row of an ECSR matrix from an OSPA. This is a routine operation on a SPA; on an OSPA, columns need to be inserted into the column array using a **heap-insert** operation.

swap Exchanges the content of an OSPA with a row of an ECSR matrix.

Given these building blocks, the overall algorithm can be composed as shown in Algorithm 3.

4 Numerical Analysis

Next, we describe the numerical behavior of the algorithm. The core numerical issue is explained in Wilkinson’s book [47, pp. 312–315]. We repeat this explanation below, for completeness, and also provide an example that shows how this numerical issue manifests itself.

The Givens variant of the algorithm is a thoroughly studied QR factorization scheme and is known to compute backward-stable factors (see [28, Section 19]). The elementary-stabilized version is not as well known, but it has also been analyzed in the literature. It has been called *pairwise pivoting* and was analyzed by Sorensen [45]. Sorensen’s analysis proves that the normwise backward error of the corresponding factorization is bounded by a product of four numbers: the magnitude of the elements of A , the unit roundoff u ($2^{-53} \approx 1.1 \times 10^{-16}$ in double-precision IEEE 754), and two growth factors. One of these growth factors represents the norm of the computed upper-triangular factor, and the other represents the norm of the implicit factor that is the equivalent of the L factor in the LU factorization with partial pivoting. In contrast with LU with partial pivoting, this implicit factor is not lower triangular, and its elements are not bounded by 1 in magnitude. Sorensen proves in his paper that the two growth factors are bounded by $2^{n-1} - 1$ and 2^{n-1} , respectively. He also notes that although these bounds are large, they are highly pessimistic, and that large growth factors have not been encountered in practice. Further experimental evidence of this was provided by Grigori, Demmel and Xiang [25, Figure 4.2].

Algorithm 3 The sparse stabilized-elementary transformations variant of the algorithm.

Input: a symmetric ECSR matrix $A \in \mathbb{R}^{n \times n}$.

Output: the number ν of negative eigenvalues of A .

```

 $\nu \leftarrow 0$ 
 $s \leftarrow$  empty OSPA of length  $n$ 
for  $i \leftarrow 1, 2, \dots, n$  do
   $x \leftarrow 0$ 
   $\text{load}(s, A_{i,:})$ 
   $(j, A_{ij}) \leftarrow \text{retrieve-head}(s)$ 
  while  $j \neq \perp$  and  $j < i$  do
    if  $|A_{jj}| < |A_{ij}|$  then
       $\text{swap}(s, A_{j,:})$ 
       $x \leftarrow x + 1$ 
      if  $\text{sign}(A_{jj}) \neq \text{sign}(A_{ij})$  then
         $x \leftarrow x + 1$ 
      end if
    end if
     $\text{subtract}(s, (A_{ij}/A_{jj}) \cdot A_{j,:})$ 
     $\text{remove-head}(s)$ 
     $(j, A_{ij}) \leftarrow \text{retrieve-head}(s)$ 
  end while
  if  $A_{ii} < 0$  then
     $x \leftarrow x + 1$ 
  end if
  if  $x \bmod 2 = 1$  then
     $\nu \leftarrow \nu + 1$ 
  end if
   $\text{store}(s, A_{i,:})$ 
end for

```

4.1 An Obstacle to a Backward-Stability Analysis

These analyses indicate that the ultimate matrix factors that we obtain from the inertia algorithm are backward stable. This may not be true if the factorization suffers from growth, but large growth is rare. The same analyses also imply backward stability for the factors that we obtain in the intermediate steps of the computation, and the same reservation about growth holds. Because the algorithm uses the factors to compute the signs of the determinant sequence

$$1, \det(A_1), \det(A_2), \dots, \det(A_n),$$

the factors' backward stability implies that each sign is backward stable. However, this does not imply the backward stability of the computed inertia. This is because each intermediate factor is modified in the steps that follow the step in which it is obtained, and therefore each sign of the determinant sequence has its own individual backward-error matrix (that equals the difference between A and a matrix \tilde{A} for which the factor is exact).

There is no reason to assume that if each factor is the exact factor of a matrix close to A , then they are all the factors of *one* matrix close to A , which would imply backward stability of the determinant sequence. Indeed, we can show that the algorithm is sometimes not backward stable.

4.2 An Example of Instability

Next, we show an example of how the algorithm can fail. We provide the transcript of a Matlab session, to make the example concrete.

We let A be a square, symmetric n -by- n matrix of the form

$$A = \begin{matrix} & \begin{matrix} n/2 & n/2 \end{matrix} \\ \begin{matrix} n/2 \\ n/2 \end{matrix} & \begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & 0 \end{bmatrix} \end{matrix}.$$

This matrix's leading diagonal block has the form

$$A_{11} = Q \operatorname{diag}(1, \epsilon_1, \epsilon_2, \dots, \epsilon_{n/2-1}) Q^T,$$

where Q is a random orthogonal matrix, and $\epsilon_1, \epsilon_2, \dots, \epsilon_{n/2-1}$ are random, independent, and normally distributed scalars with mean 0 and a standard deviation that equals the unit roundoff. The notation $\operatorname{diag}(\cdot)$ indicates a diagonal matrix with the specified diagonal elements. The block A_{21} is defined so that its elements are random, independent, and normally distributed with mean 0 and standard deviation 1.

In Matlab, we construct this matrix as follows:

```
>> n = 2048;
>> Q = orth(randn(n / 2));
>> d = [1; eps * randn(n / 2 - 1, 1)];
>> A11 = Q * diag(d) * Q';
>> A21 = randn(n / 2);
>> A = [A11 A21'; A21 zeros(n / 2)];
```


Next, we compute A 's eigenvalues by running the Matlab command `eig`. Using these eigenvalues, we compute the matrix's condition number and its negative index of inertia:

```
>> ev = eig(A);
>> kappa = max(abs(ev)) / min(abs(ev))
kappa = 3.4498e+03

>> nu = sum(ev < 0)
nu = 1024
```

In this example, the `eig` command calls LAPACK's symmetric eigensolver [2], which is backward stable. Its backward stability guarantees that the computed eigenvalues, condition number, and negative index of inertia are those of some matrix \tilde{A} close to A . As we see above, the condition number is small, and so all of the matrices in the vicinity of \tilde{A} , including A itself, have eigenvalues with the same signs as \tilde{A} . This means that all such matrices have the same negative index of inertia, namely 1,024.

Calling Wilkinson's algorithm, we obtain a different result:

```
>> our_nu = inertia(A)
our_nu = 1026
```

This shows that the algorithm is not backward stable. Any backward-stable algorithm would return the negative index of inertia of a matrix in the vicinity of A and \tilde{A} , which must be 1,024.

The result is similar regardless of whether we use the Givens variant of the algorithm or the elementary-stabilized one. In particular, because growth is not possible in the Givens variant, this shows that the instability is not caused by growth.

4.3 Instability Is Caused by Near-Singularity

The cause of the instability is singularity of the leading principal minors. In our example, all of the matrix's leading principal submatrices have a condition number of order u^{-1} , except for the three submatrices of orders 1, $n - 1$ and n . Because the other $n - 3$ submatrices are nearly singular, the computed signs of their determinants are often incorrect, even when computed using a backward-stable algorithm, and therefore counting the number of sign changes in the determinant sequence yields an incorrect result.

4.4 The (Limited) Effect of the Instability on Bisection

In the experiments that we describe in the next section, the eigenvalues that we produce using bisection are usually as accurate as those that we produce using a backward-stable eigensolver. The instability of the inertia algorithm does not appear to have an effect on bisection.

The reason for this is that the instability manifests itself only when $A - xI$ has a leading principal submatrix that is nearly singular. To make one of these submatrices nearly singular, x must be at a distance of $O(u)\|A\|$ from an eigenvalue of such a submatrix. This can happen under two scenarios. First, as bisection converges to an eigenvalue of A , the point x moves progressively closer to that eigenvalue until it reaches within $O(u)\|A\|$. This scenario does not create a problem, because when x is close to an eigenvalue of A , an error in the computed inertia cannot prevent bisection from converging [47, pp. 302–305]. The other scenario is that x is close to an eigenvalue of a leading principal submatrix of A , and that eigenvalue is far from every eigenvalue of A . This can lead bisection to fail. However, such unfortunate positioning of x appears to happen only in pathological cases, meaning that the instability has few opportunities to have an effect.

5 Numerical Experiments

We carried out extensive numerical experiments to test the accuracy of the algorithm. In these experiments, we used our inertia algorithm as part of a bisection eigensolver to compute the eigenvalues of a variety of matrices. Although in practice bisection is often not a competitive algorithm, it requires a large number of inertia computations, and this makes it a convenient platform to stress-test our algorithm.

We computed the eigenvalues of two families of matrices. The first family is one that we produced using the LAPACK subroutine LATMS, which generates random dense matrices for testing LAPACK subroutines. These matrices have the form $A = Q\Lambda Q^T$, where Q is a random orthogonal matrix and Λ is a diagonal matrix whose diagonal elements $\lambda_1, \lambda_2, \dots, \lambda_n$ are the eigenvalues of A . The eigenvalues are randomly distributed according to two parameters named **mode** and κ . There are six possible modes, in the first five of which the eigenvalues have the form $\lambda_i = s_i \sigma_i$, where the scalars s_i take the values ± 1 independently with equal probability, and the scalars σ_i have a distribution that depends on the mode. In all of the distributions that correspond to the first five modes, the σ_i are nonnegative, and therefore they are the singular values of A . The distributions are the following:

1. The first singular value is equal to 1, and all others are equal to $1/\kappa$.
2. The first $n - 1$ singular values are equal to 1, and $\sigma_n = 1/\kappa$.
3. The singular values are evenly distributed between 1 and $1/\kappa$, on a logarithmic scale.
4. The singular values are evenly distributed between 1 and $1/\kappa$, on a linear scale.
5. The singular values have the form $(1/\kappa)^e$, where e is drawn independently for each singular value from the uniform distribution on the interval $(0, 1)$.

Table 1: Accuracy of the computed eigenvalues of random dense matrices, generated using the LAPACK subroutine LATMS.

mode	maximal error				
	$\kappa = 10^1$	$\kappa = 10^4$	$\kappa = 10^8$	$\kappa = 10^{12}$	$\kappa = 10^{16}$
1	7.7×10^{-16}	7.0×10^{-16}	4.9×10^{-16}	2.1×10^{-16}	1.7×10^{-15}
2	9.0×10^{-16}	7.5×10^{-16}	1.9×10^{-15}	5.3×10^{-16}	1.8×10^{-15}
3	1.7×10^{-15}	1.8×10^{-15}	2.1×10^{-15}	1.7×10^{-15}	1.8×10^{-15}
4	1.8×10^{-15}	1.8×10^{-15}	1.8×10^{-15}	1.8×10^{-15}	1.8×10^{-15}
5	1.8×10^{-15}	1.8×10^{-15}	1.8×10^{-15}	1.8×10^{-15}	1.8×10^{-15}
6 ^{a,b}	1.8×10^{-15}	1.8×10^{-15}	1.8×10^{-15}	1.8×10^{-15}	1.8×10^{-15}

^a Matrices generated in mode 6 do not depend on κ , and therefore all of the entries in the bottom row represent the same matrix.

^b The frequent appearance of the number 1.8×10^{-15} in the first five rows is an artifact of rounding to two significant digits; actual numbers are all distinct.

Finally, in mode 6, LATMS generates the λ_i directly without generating the singular values first. In this mode, the λ_i are independent and normally distributed with mean 0 and standard deviation 1.

We generated one matrix of order $n = 256$ for each combination of **mode** and κ , for various values of κ , and computed all of the eigenvalues of that matrix. We then measured their accuracy using the maximal normalized error,

$$\max_{i=1,2,\dots,n} |\lambda_i - \hat{\lambda}_i| / \|A\|_1,$$

where the λ_i are the actual eigenvalues used by LATMS and the $\hat{\lambda}_i$ are the approximate ones produced by our bisection eigensolver. The full results are shown in Table 1. All of the errors are within one order of magnitude from $u \approx 1.1 \times 10^{-16}$.

The second family of matrices in our experiments are sparse matrices from the University of Florida Sparse Matrix Collection [9]. We computed the inertia of all 283 real symmetric matrices of order $64 < n < 16,384$ in this collection whose elements have known numerical values. If the running time $T(A)$ of our algorithm satisfied $53nT(A) \leq 2$ hours, where n is the order of A , we ran our bisection algorithm to compute all of the eigenvalues of A . If $T(A)$ was higher, indicating that the eigensolver would take more than 2 hours to run (it needs 53 inertia computations per eigenvalue), we did not compute the eigenvalues. This selection process left us with 116 matrices, described below in Table 2.

As in the previous experiment, we computed the eigenvalues of each of the 116 matrices, and measured the maximal normalized error (relative to backward-stable eigenvalues computed using the LAPACK subroutine SYEV). The

Table 2: Statistical parameters of the set of University of Florida matrices.

	n	nnz/n	κ
minimum	66.0	0.5	5.0
1st quartile	380.0	3.8	7.3×10^3
median	958.5	6.9	1.6×10^6
3rd quartile	1920.5	16.8	7.5×10^{10}
maximum	15,439.0	97.3	inf

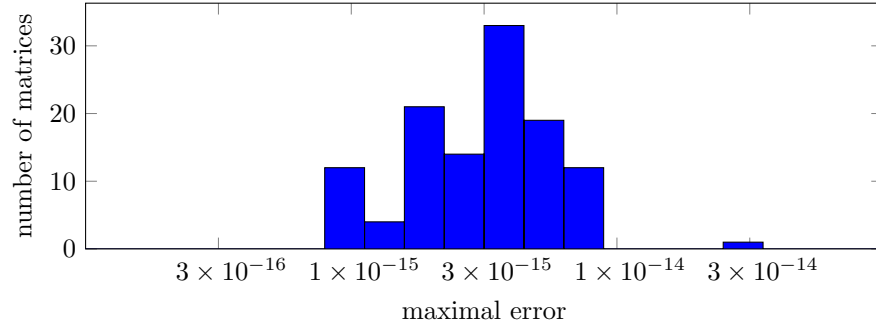


Figure 2: Accuracy of the computed eigenvalues of sparse matrices from the University of Florida collection.

results are shown in a histogram plot in Figure 2. The errors are again comparable to u , ranging between 8.0×10^{-16} and 3.5×10^{-14} , with a median of 3.5×10^{-15} .

6 Performance Experiments

In this section, we describe the experiments that we carried out to study the performance of our sparse inertia algorithm. We analyzed the performance of the two variants of our algorithm, and we compared it with the performance of two other factorization algorithms. The first algorithm that we compared ourselves with is SuiteSparseQR, or SPQR for short [8]. It is a state-of-the-art sparse QR factorization algorithm, and it is the algorithm that runs when we apply Matlab's `qr` command to a sparse matrix. Our interest in SPQR stems from the fact that it computes the same factorization that we compute in the Givens variant of our algorithm. Comparing ourselves with SPQR helps us to learn how much performance we can gain by adopting the advanced techniques that are implemented in SPQR, such as the use of supernodes.

The second algorithm with which we compared ourselves is the algorithm

MA57 from the HSL library [15, 29]. It is an industrial-quality sparse-factorization algorithm for symmetric-indefinite matrices. An important distinction between SPQR and MA57 is that the former does not compute the inertia, while the latter does. In these experiments, MA57 represents the alternative to our approach to the problem of computing the inertia. A comparison with MA57 allows us to gain insight into the relative strengths and weaknesses of our algorithm.

We carried out two experiments on a single core of an Intel i7-2600 CPU, using Matlab R2012b, which is bundled with Intel MKL BLAS, version 10.3.9, and SuiteSparseQR version 1.1.0. We used version 3.8.0 of MA57, version 5.1.0 of the graph partitioning library METIS [33], and version 5.3.0 of the PAPI library [40], which we used for counting floating-point operations. We compiled our code and that of MA57 using version 13.1.1 of the Intel compiler suite.

Our first experiment was carried out on a set of 81 matrices from the University of Florida Sparse Matrix Collection. The matrices were chosen by selecting all of the symmetric, real, indefinite matrices of order $32,768 \leq n \leq 65,536$. We limited the computation with each matrix to one hour of time and 16 GB of memory. Twenty-one of the matrices exceeded these limits and were therefore removed from the experiment; this left us with a set of 60 matrices.

For each matrix, we computed a nested-dissection ordering for A using METIS, and then applied each of the algorithms to the reordered matrix. Figure 3 compares the performance of the Givens variant of our algorithm to that of SPQR. We find that the required number of arithmetic operations (flops) in the two algorithms is within a factor of 10 of each other, with no significant advantage in favor of either one. The reason for the variation in flops is because the two algorithms process the rows in different orders, and the order of the rows can have a dramatic effect on flops. This has already been noted by George and Heath [18, p. 78]. The computational rate of SPQR is however much higher, ranging between 2.5 and 15 Gflop/s, whereas that of our algorithm ranges between 474 and 717 Mflop/s. This also translates to a dramatic difference in running times in favor of SPQR.

Figure 4 shows that the elementary-stabilized variant of the algorithm is much more efficient than the Givens variant. The elementary-stabilized variant performs a factor of 0.001 to 0.13 of the flops that the Givens variant performs. The computational rates of the two codes are comparable, with more variability in the elementary-stabilized case because of row exchanges. The elementary-stabilized version is therefore faster by a significant factor, between 5.95 and 366.

Figure 5 compares the elementary-stabilized variant with SPQR. Although, as we have seen in the previous figures, the computational rate of our algorithm is orders of magnitude lower, we preserve sparsity much better. Our algorithm always performs fewer flops than SPQR, with a median ratio of 0.017. Due to the much better sparsity, we are actually faster in 45 of the 60 matrices, with a time ratio that ranges between 0.0011 and 3.51.

Finally, Figure 6 compares the performance of the elementary-stabilized variant with MA57. We see that MA57 is better than our algorithm at preserving sparsity, requiring up to a factor of 21.7 fewer flops than our algorithm. MA57

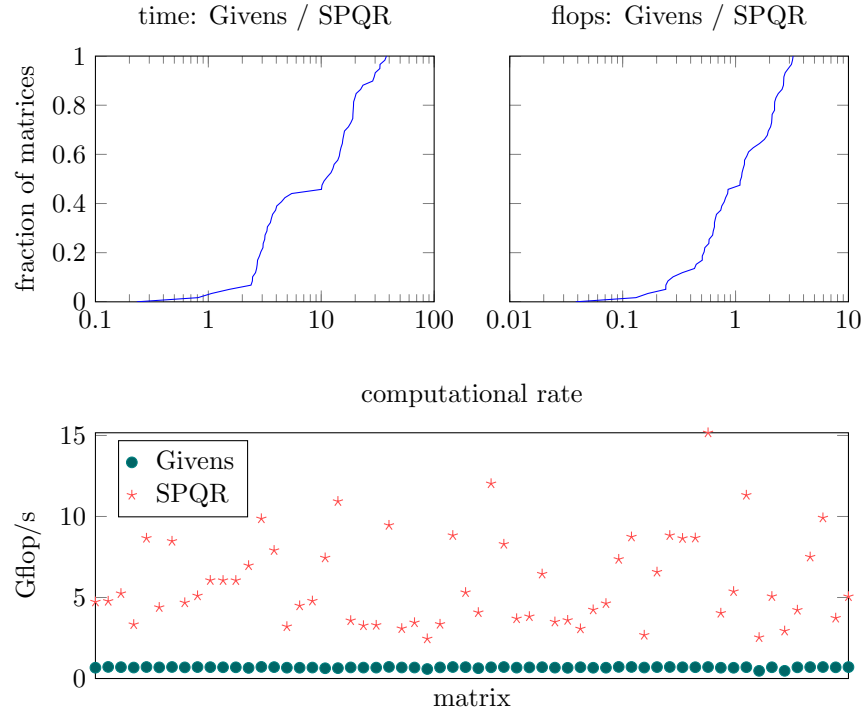


Figure 3: A comparison of the Givens variant of our algorithm with SPQR. The top plots show two empirical cumulative-distribution-function curves of the ratio of the running time of our algorithm to that of SPQR (left) and the ratio of the numbers of arithmetic operations of the two algorithms (right). The bottom plot shows the computational rates of the two algorithms.

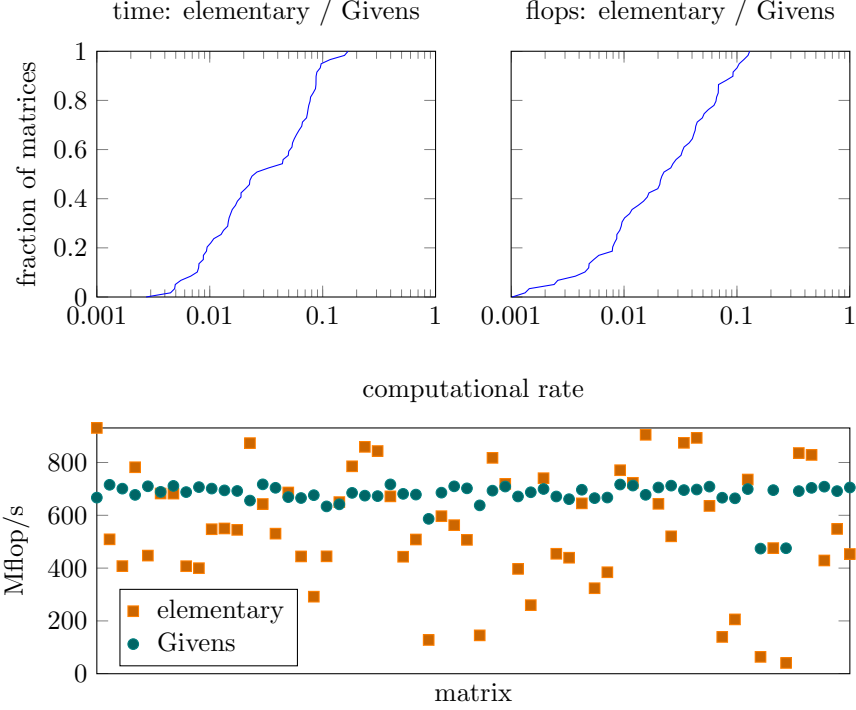


Figure 4: A comparison of the two variants of our algorithm using the same type of plots as in Figure 3.

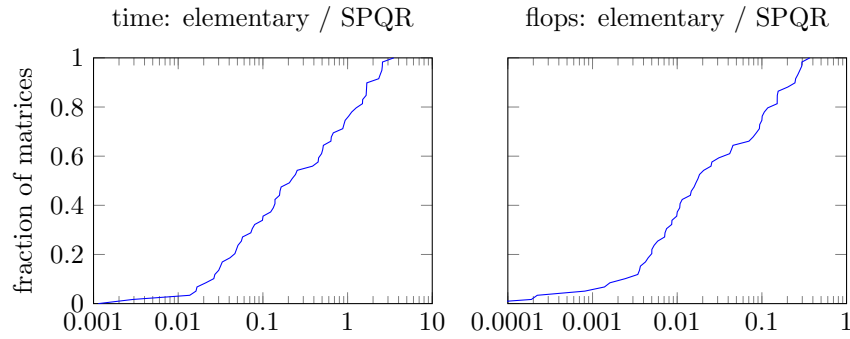


Figure 5: A comparison of the elementary-stabilized variant of our algorithm with SPQR. (The computational rates of the two algorithms are presented in Figures 3 and 4.)

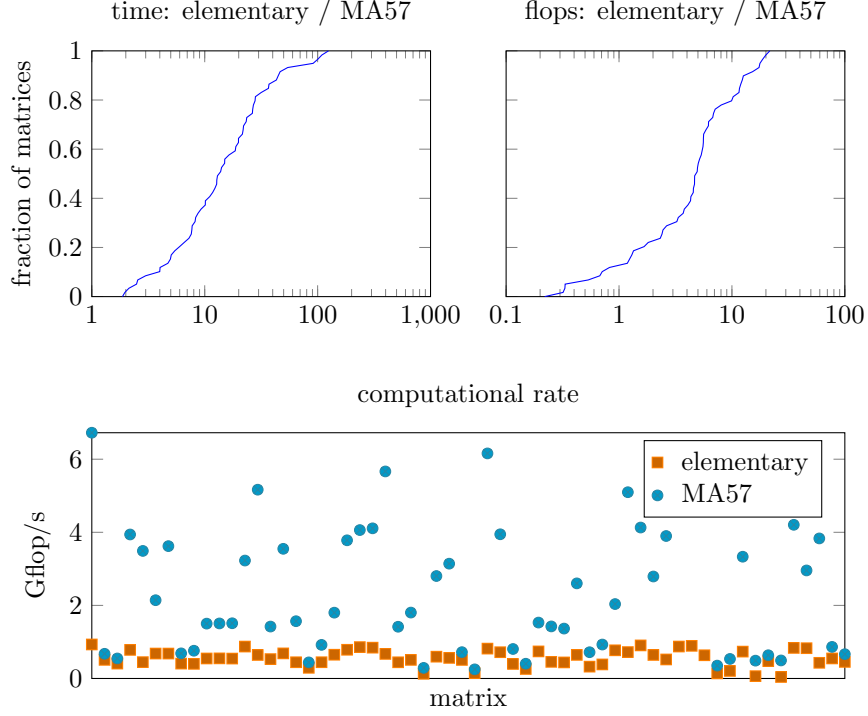


Figure 6: A comparison of the elementary-stabilized variant of our algorithm with MA57.

also achieves a significantly faster computational rate of up to 6.72 Gflop/s, although in 20 of the matrices it falls below 1 Gflop/s and can be as low as 250 Mflop/s. Nevertheless, MA57 is always faster, by a factor that ranges between 1.87 and 124.8.

In our final experiment, we made a more detailed comparison of our algorithm with MA57. Our goal is to find matrices where our more conservative approach pays off. We started with the 422 symmetric, real, indefinite matrices of order $8,192 \leq n \leq 1,048,576$ in the University of Florida collection, but kept only the 54 matrices whose MA57 factor had at least twice as many nonzeros as the Cholesky factor would have contained (had the matrix been positive definite). This produced a set of matrices on which pivoting in MA57 caused significant fill. For each of the 54 matrices, we invoked our factorization and MA57 using two orderings: an optimistic nested-dissection ordering of A and a conservative nested-dissection ordering of the structure of $A^T A$. We then compared MA57 and our algorithm with respect to a parameter we refer to as fill, which we define as the number of nonzeros in the upper-triangular factor, divided by the number of nonzeros in A . In 19 of the matrices we found that the fill in our algorithm is lower than the fill of MA57, sometimes dramatically

so. A list of these matrices is given in Table 3.

7 Discussion and Conclusions

In this paper, we proposed a sparse variant of Wilkinson’s inertia algorithm. The new inertia algorithm provides provable a priori bounds on the fill, which other algorithms do not provide.

Our performance experiments show that even without using the techniques that are necessary to obtain top performance from a sparse-matrix algorithm (such as using supernodes), our algorithm is competitive and often superior to a top-quality sparse QR factorization code due to the sparser factors that we produce in our algorithm.

Nevertheless, our algorithm is usually slower than MA57, often much slower. Due to the use of supernodes in MA57, it achieves much higher computational rates, and thanks to its use of threshold pivoting and a matching algorithm that allows it to rescale and reorder the matrix so as to prevent pivoting [16], it is better able to preserve sparsity. These techniques can be used in our algorithm as well, and we expect significant improvement if they are used. This is left for future work. Even without this, we found several cases where our approach yields less fill, and one matrix (Andrianov/net150) that MA57 cannot factor within the allotted time and space, whereas we can.

Furthermore, we also found that in some matrices, the computational rate of MA57 can be quite low, lower than that of our algorithm. It has been noted by Davis and Palamadai Natarajan [10] that extremely sparse matrices that do not suffer from significant fill can benefit from using an algorithm that does not use supernodes. We believe that this is the type of matrices where MA57 suffers from poor performance, and where our code can offer a good alternative.

Acknowledgments This research was supported in part by grant 1045/09 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities) and by grant 2010231 from the US-Israel Binational Science Foundation.

References

- [1] Hasan Metin Aktulga, Lin Lin, Christopher Haine, Esmond G. Ng, and Chao Yang. Parallel eigenvalue calculation based on multiple shift-invert Lanczos and contour integral based spectral projection method. *Parallel Computing*, 40(7):195–212, 2014.
- [2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users’ Guide*. SIAM, 3rd edition, 1999.

Table 3: Matrices where our algorithm produced less fill than MA57. We define fill as the number of nonzero elements in the triangular factor, divided by the number of nonzeros in A . Column headings nd and wide indicate nested-dissection ordering of A and of $A^T A$ respectively (the latter of the two is equivalent to a wide-separator nested-dissection ordering of A). The symbol \dagger indicates that the computation exceeded the maximal allocated time of one hour, and \ddagger indicates that the algorithm required more memory than the allocated 16 GB. MA57 always produced more fill with a nested-dissection ordering of the structure of $A^T A$ than with the structure of A , so these results are not shown.

matrix name	n	nnz/n	fill		
			MA57	our algorithm	
				nd	wide
AG-Monien/3elt_dual	9,000	3.0	9.5	11.4	6.9
AG-Monien/big_dual	30,269	3.0	11.0	13.8	8.2
AG-Monien/crack_dual	20,141	3.0	8.3	11.1	7.4
AG-Monien/L-9	17,983	4.0	14.2	13.9	10.7
AG-Monien/whitaker3_dual	19,190	3.0	11.9	16.9	10.1
Andrianov/net150	43,520	71.7	\ddagger	34.0	\dagger
Andrianov/pattern1	19,242	484.5	7.2	5.6	9.4
DIMACS10/t60k	60,005	3.0	12.8	20.1	10.9
GHS_indef/dtoc	24,993	2.8	4.0	3.4	1.8
Gleich/usroads	129,164	2.6	5.5	4.9	4.0
Gleich/usroads-48	126,146	2.6	5.5	4.9	4.1
Gset/G66	9,000	4.0	16.3	26.3	14.4
Gset/G67	10,000	4.0	16.0	27.1	14.2
Gupta/gupta3	16,783	555.5	6.6	5.0	11.5
Rajat/rajat06	10,922	4.3	6.2	5.0	3.8
Rajat/rajat07	14,842	4.3	6.3	5.6	4.0
Rajat/rajat08	19,362	4.3	6.5	5.6	4.3
Rajat/rajat09	24,482	4.3	6.7	6.4	4.5
Rajat/rajat10	30,202	4.3	6.6	6.3	4.8

- [3] Igor Brainman and Sivan Toledo. Nested-dissection orderings for sparse LU with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 23(4):998–1012, 2002.
- [4] D. Calvetti, L. Reichel, and D. C. Sorensen. An implicitly restarted Lanczos method for large symmetric eigenvalue problems. *Electronic Transactions on Numerical Analysis*, 2:1–21, 1994.
- [5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.
- [6] Timothy A. Davis. Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):196–199, June 2004.
- [7] Timothy A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, 2006.
- [8] Timothy A. Davis. Algorithm 915, SuiteSparseQR: Multifrontal multi-threaded rank-revealing sparse QR factorization. *ACM Transactions on Mathematical Software*, 38(1):8:1–8:22, 2011.
- [9] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38:1:1–1:25, 2011.
- [10] Timothy A. Davis and Ekanathan Palamadai Natarajan. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Transactions on Mathematical Software*, 37(3):36:1–36:17, September 2010.
- [11] James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.
- [12] James W. Demmel and William Gragg. On computing accurate singular values and eigenvalues of matrices with acyclic graphs. *Linear Algebra and Its Applications*, 185:203–217, 1993.
- [13] Inderjit S. Dhillon, Beresford N. Parlett, and Christof Vömel. The design and implementation of the MRRR algorithm. *ACM Transactions on Mathematical Software*, 32:533–560, 2006.
- [14] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, 9(3):302–325, 1983.
- [15] Iain S. Duff. MA57—a code for the solution of sparse symmetric definite and indefinite systems. *ACM Transactions on Mathematical Software*, 30:118–144, 2004.
- [16] Iain S. Duff and Stéphane Pralet. Strategies for scaling and pivoting for sparse symmetric indefinite problems. *SIAM Journal on Matrix Analysis and Applications*, 27(2):313–340, 2005.

- [17] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.
- [18] Alan George and Michael T. Heath. Solution of sparse linear least squares problems using Givens rotations. *Linear Algebra and Its Applications*, 34:69–83, 1980.
- [19] Alan George and Esmond Ng. On the complexity of sparse QR and LU factorization of finite-element matrices. *SIAM Journal on Scientific and Statistical Computing*, 9(5):849–861, 1988.
- [20] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.
- [21] John R. Gilbert, Esmond G. Ng, and Barry W. Peyton. An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 15(4):1075–1091, 1994.
- [22] John R. Gilbert, William W. Pugh, Jr., and Tatiana Shpeisman. Ordered sparse accumulator and its use in efficient sparse matrix computation. US Patent 5,983,230, 1999.
- [23] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 4th edition, 2013.
- [24] Laura Grigori, Erik G. Boman, Simplicio Donfack, and Timothy A. Davis. Hypergraph-based unsymmetric nested dissection ordering for sparse LU factorization. *SIAM Journal on Scientific Computing*, 32(6):3426–3446, 2010.
- [25] Laura Grigori, James W. Demmel, and Hua Xiang. CALU: A communication optimal LU factorization algorithm. *SIAM Journal on Matrix Analysis and Applications*, 32(4):1317–1350, 2011.
- [26] K. K. Gupta. Free vibrations of single-branch structural systems. *Journal of the Institute of Mathematics and its Applications*, 5(3):351–362, 1969.
- [27] K. K. Gupta. Vibration of frames and other structures with banded stiffness matrix. *International Journal for Numerical Methods in Engineering*, 2(2):221–228, 1970.
- [28] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2nd edition, 2002.
- [29] HSL, a collection of Fortran codes for large-scale scientific computation. See <http://www.hsl.rl.ac.uk/>.
- [30] Yifan Hu and Jennifer Scott. Ordering techniques for singly bordered block diagonal forms for unsymmetric parallel sparse direct solvers. *Numerical Linear Algebra with Applications*, 12(9):877–894, 2005.

- [31] Jochen A. G. Jess and H. G. M. Kees. A data structure for parallel L/U decomposition. *IEEE Transactions on Computers*, C-31(3):231–239, March 1982.
- [32] W. Kahan. Accurate eigenvalues of a symmetric tri-diagonal matrix. Technical Report CS41, Computer Science Department, Stanford University, 1966.
- [33] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20:359–392, 1998.
- [34] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK Users' Guide*. SIAM, 1997.
- [35] Xiaoye S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software*, 31(3):302–325, September 2005.
- [36] Richard J. Lipton, Donald J. Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16(2):346–358, 1979.
- [37] Joseph W. H. Liu. On general row merging schemes for sparse Givens transformations. *SIAM Journal on Scientific Computing*, 7(4):1190–1211, 1986.
- [38] Joseph W. H. Liu, Esmond G. Ng, and Barry W. Peyton. On finding supernodes for sparse matrix computations. *SIAM Journal on Matrix Analysis and Applications*, 14(1):242–252, 1993.
- [39] R. S. Martin and J. H. Wilkinson. Solution of symmetric and unsymmetric band equations and the calculation of eigenvectors of band matrices. *Numerische Mathematik*, 9(4):279–301, 1967.
- [40] PAPI: Performance application programming interface. Innovative Computing Laboratory, University of Tennessee.
- [41] Beresford N. Parlett. *The Symmetric Eigenvalue Problem*. SIAM, 1998.
- [42] Donald J. Rose, R. Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976.
- [43] Olaf Schenk and Klaus Gärtner. On fast factorization pivoting methods for sparse symmetric indefinite systems. *Electronic Transactions on Numerical Analysis*, 23:158–179, 2006.
- [44] David S. Scott. Computing a few eigenvalues and eigenvectors of a symmetric band matrix. *SIAM Journal on Scientific Computing*, 5(3):658–666, 1984.

- [45] Danny C. Sorensen. Analysis of pairwise pivoting in Gaussian elimination. *IEEE Transactions on Computers*, C-34:274–278, 1985.
- [46] G. W. Stewart. *Matrix Algorithms: Volume II: Eigensystems*. SIAM, 2001.
- [47] J. H. Wilkinson. *Algebraic Eigenvalue Problem*. Clarendon Press, Oxford, 1965.
- [48] Hong Zhang, Barry Smith, Michael Sternberg, and Peter Zapol. SIPs: Shift-and-invert parallel spectral transformations. *ACM Transactions on Mathematical Software*, 33(2), 2007.